The True Online Continuous Learning Automation (TOCLA) in a continuous control benchmarking of actor-critic algorithms

https://www.overleaf.com/project/5f68dc0f92bed70001d535cc

The True Online Continuous Learning Automation (TOCLA) in a continuous control benchmarking of actor-critic algorithms

Gordon Frost
The School of Engineering and Physical Sciences
Heriot-Watt University
Edinburgh, UK
https://orcid.org/0000-0001-6908-3738

Marta Vallejo
The School of Engineering and Physical Sciences
Heriot-Watt University
Edinburgh, UK
m.vallejo@hw.ac.uk

***Abstract*—Reinforcement learning problems are often discretised, use linear function approximation, or perform batch updates. However, many applications that can benefit from reinforcement learning contain continuous variables and are inherently non-linear, for example, the control of aerospace or maritime robotic vehicles. Recent work has brought focus onto online temporal difference methods, specifically for using non-linear function approximation. In this paper, we evaluate the Forward Actor-Critic against the regular Actor-Critic, and Continuous Actor-Critic Learning Automation. We also propose and evaluate a new algorithm called True Online Continuous Learning Automation (TOCLA) which combines these two approaches. The chosen benchmark problem was the MountainCarContinuous-v0 environment from OpenAI Gym, which represents a further step in complexity over the benchmark used to test the Forward Actor Critic in previous works. Our results demonstrate the superiority of TOCLA in terms of its sensitivity to hyper-parameter selection compared with the Forward Actor Critic, Continuous Actor-Critic Learning Automation, and Actor Critic algorithms.**

***Index Terms*—Reinforcement Learning, Actor-Critic, TOCLA, CACLA, Forward Actor-Critic, Nonlinear Function Approximation.**

I. Introduction

Real-world problems such as modelling the dynamics of an aircraft or underwater vehicle are non-linear and inherently complex [1]. In many cases, these problems use a simplified model representing the system dynamics, since a complete model is simply not known or is too computationally expensive [2]. For these applications, a model-free control strategy is desirable, as it removes the need for a complex or potentially inaccurate model. Whilst being model-free, a solution must also be capable of handling the continuous variables observed in real-world systems. One way of achieving this is to learn the control strategy directly from interacting with the continuous environment, as it is defined in a Markov Decision Process (MDP). The Reinforcement Learning (RL) paradigm does exactly this – from interacting with the real control process, the agent learns the optimal sequence of actions to execute that will maximise an objective (accumulated reward). RL contains two fields that are of particular interest for such real-world problems: Actor-Critic (AC) models, and Temporal Difference (TD) learning [3].

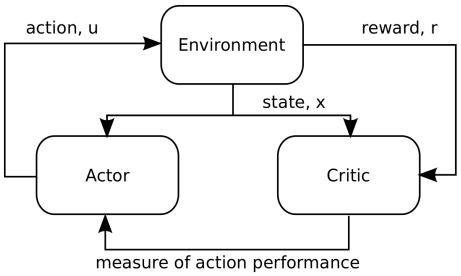

Fig. 1: The Actor-Critic model.

AC models combine the convergence guarantees of policy-gradient strategies with the interaction sample efficiency of value-function approaches. As it can be seen in Fig.1, at each timestep of the MDP, an AC model interacts with its environment, first, by taking an action sampled from the actor's approximator, which moves the learning agent from the current state to a new state, where it receives a reward. The actor represents the policy, which is a function that maps a given state to the action that maximises the future reward. The critic contains a value-function, mapping the state to the expected future reward from that state. The reward is used to update the critic value-function, which generates state estimates. The actor is, then, updated based on the state-estimate of the current and previous state. By storing these functions in separate parameterised approximators, continuous state and action spaces can easily be supported and the variance of the policy-gradient actor can also be reduced by the bias of the critic.

TD learning aims to improve a state value estimate, by also considering the value of temporally nearby states. To achieve this, a one or multi-step update target is used to update the function approximator. One of the most popular and well-known TD algorithms is TD(λ) [4], used as the critic in the

standard AC algorithm. However, van Seijen *et al.* [5] showed that TD(λ) only approximates the update targets defined by the λ-return algorithm when updating the value estimates online. The Forward TD(λ) algorithm, proposed by van Seijen [6], was designed to overcome this problem by using multi-step update targets and non-linear function approximation methods such as Artificial Neural Networks (ANNs). Veeriah *et al.* [7] then incorporated Forward TD(λ) into an AC model called Forward AC (FAC).

The Continuous Actor-Critic Learning Automation (CACLA) algorithm [8] is based on the original AC algorithm, but with a modified actor update rule. This modification addresses some issues of the AC algorithm policy update and makes the algorithm more suited to continuous control applications [9]. One such benefit of the CACLA actor update is that it is invariant to the scale of the reward function.

TOCLA is a new approach that uses the Forward TD(λ) algorithm as the critic, and the actor update rule from CACLA. Evaluating these four algorithms on the same benchmark (a more representative benchmark problem closer to typical real-world applications) provides a quantitative assessment of the benefit of the Forward TD(λ) and CACLA actor updates, and hence, TOCLA, which combines these.

In this paper, to the best of the authors' knowledge, we present a novel quantitative comparison of these four TD AC algorithms: AC, CACLA, Forward AC, and TOCLA.

The benchmark selected is the OpenAI Gym MountainCarContinuous-v0 environment [10], which represents a further step in complexity from the discrete-action mountain car environment used in Veeriah *et al.* [7]. This problem is harder to solve for two reasons: firstly, the action is a single-dimension continuous value, which represents the force excerpted on the car; secondly, the objective for this environment is for the car to reach the goal state, whilst minimising the excerpted force. Hence, the reward function is designed to penalise large action magnitudes. This objective formulation effectively penalises "bang-bang" control strategies (which can perform well on the discrete action mountain car environment) and represents a more desirable optimal control strategy for real-world applications such as robotics.

We also address the challenge of selecting optimal algorithm hyperparameters through the use and development of a Genetic Algorithm (GA) library. GA is a population-based optimisation algorithm that has been used extensively for hyperparameter optimisation [11]. In this case, GA is designed to optimise the actor and critic learning rates and the λ parameter, as these are critical to the algorithm's performance [12]. The proposed GA library is called GEneralised Genetic Algorithm (GEGA) and we make it openly available in GitHub [13]. GEGA aims to be a flexible general-purpose implementation to suit many optimisation problems.

Five contributions are presented in this paper, where we:

A) propose a new AC algorithm called TOCLA, that uses the Forward TD(λ) algorithm as the critic to estimate state values. To update the actor, the typical update rule is exchanged for the CACLA update rule due to its invariance to the reward scaling.

B) provide the wider research community with a repeatable python implementation of the AC, CACLA, FAC, and TOCLA algorithms, implemented as an extension to the comprehensive framework called the Autonomous Learning Library (ALL) [14], available in GitHub repository [15].

C) go one step further with Forward AC than Veeriah *et al.* [7] and observe its performance in comparison to similar TD AC algorithms, when applied to a control environment that has a continuous action space. Results are presented for the MountainCarContinuous-v0 OpenAI Gym environment.

D) publish a flexible GA implementation called the GEneralised Genetic Algorithm (GEGA) that can be used for optimisation of a RL agent's hyperparameters, available in GitHub repository [13].

E) publish the optimal hyper-parameter values obtained for the compared algorithms on the OpenAI Gym MountainCarContinuous-v0 benchmark.

The rest of this paper is structured as follows: Section II introduces the related work and concepts useful in understanding the algorithms of this paper. Section III describes the studied algorithms and their major differences. A generalised view of the components and tools used in this work is presented in Section IV, and exact details of how these were used are shown in Section V. Section VI describes the results we obtained, before discussing their meaning in Section VII. Finally, we make concluding statements and highlight potential future work in Section VIII.

II. Related Work

A *"forward view"* exists for the TD(λ) algorithm, which provides an intuitive way to understand how TD methods learn [3]. The *"forward view"* aims to move a state estimate towards the λ-return update target. Shown in Equ.1, $G_t^\lambda(\theta)$ is defined as an infinite weighted sum over n-step returns, where the n-step return is defined in Equ.2 as an update target based on the rewards gained over n time steps into the future.

$$G_t^\lambda = (1-\lambda)\sum_{i=0}^{\infty}\lambda^{i-1}G_t^{(i)} \tag{1}$$

$$G_t^{(n)} = \sum_{j=1}^{n}\gamma^{j-1}r_{t+j} + \gamma^n\hat{V}(x_{t+n}) \tag{2}$$

where G_t^λ is the λ-return, $G_t^{(n)}$ is the n-step return, γ is the discount factor, r_t is the reward received at timestep t, λ is the trace weight for multi-step estimates, x_t is the MDP state at time t, and $\hat{V}(x_{t+n})$ is the expected return for being in the state n steps in the future.

A trade-off between bias (one-step) and variance (multi-step) of the update target can be managed using the λ parameter. However, Seijen and Sutton demonstrated in [5] and [6] that the TD(λ) algorithm only approximates the λ-return when state estimates are updated online. To achieve exact equivalence to the *"forward view"* of TD learning, they

proposed the 'Online λ-return' algorithm, where True Online TD(λ) exactly matches this new *"forward view"* when state estimates are updated online.

The True Online TD(λ) algorithm was shown to empirically out-perform TD(λ) and some control variants on a random Markov process, a real-world myoelectric prothestic arm, a random walk task, and a mountaincar benchmark problem. However, all these algorithms and results thus far have only considered the use of linear function approximation, since non-linearity can easily lead to divergence [16], [17]. In [6], the True Online TD(λ) algorithm and its *"forward view"* was extended to create the Forward TD(λ) and K-bounded λ-return algorithms - a backward and *"forward view"*, specifically designed to exactly match when using non-linear function approximators. K truncates the infinite sum to the first K n-step returns, allowing it to be computed online with a small delay. The algorithm was validated on the mountaincar and cartpole benchmark problems, with the action space discretised using a separate ANN for each of the two actions' value functions. Veeriah *et al.* [7] furthered the work on Forward TD(λ), by incorporating it into an AC model, creating the Forward Actor Critic (FAC) algorithm. Its performance was, again, presented on the mountaincar and cartpole benchmark environments, where the actor output was an array of the probability for each discrete action.

III. Actor Critic Algorithms

The AC algorithm forms the basis for the three AC variants compared in this paper, namely FAC, CACLA, and TOCLA. Table I summarises the differences between the evaluated ACs, showing their actor update and critic algorithm. In the AC algorithm, the critic using TD(λ) moves the state estimates towards the λ-return update target (Equ.1), which is then used to compute the Temporal Difference Error (TDE) (Equ.3). The TDE represents whether the action chosen at the previous timestep led to a better or worse state than expected.

$$\delta_k = G_t^\lambda - V_{\theta_k}(x_k) \tag{3}$$

where δ_k is the TDE, G_t^λ is the λ-return update target, and $V_{\theta_k}(x_k)$ is the state estimate (given by the critic) for state, x_k, parameterised by the vector $\theta \in \mathbb{R}^m$.

Then, the critic is updated by Equ.4, as follows:

$$\theta_{k+1} = \theta_k + \alpha_{c,k}\delta_k \nabla_\theta V_{\theta_k}(x_k) \tag{4}$$

where $\nabla_\theta V_{\theta_k}(x_k)$ is the gradient of the state value function, and α_c is the critic's learning rate.

The actor's approximator is parameterised by the vector $\psi \in \mathbb{R}^n$ and is updated using Equ.5.

$$\psi_{k+1} = \psi_k + \alpha_\pi \delta_k \frac{\nabla_\psi \pi(u_t|x_t,\psi_t)}{\pi(u_t|x_t,\psi_t)} \tag{5}$$

where α_π is the policy's learning rate, and $\frac{\nabla_\psi \pi(u_t|x_t,\psi_t)}{\pi(u_t|x_t,\psi_t)}$ is the policy gradient [18].

Core differences among AC algorithms

Algorithm	Actor	Critic	Reference
AC	TDE	TD(λ)	[3]
FAC	TDE	Forward TD(λ)	[7]
CACLA	Exploration	TD(λ)	[8]
TOCLA	Exploration	Forward TD(λ)	Approach proposed

TABLE I: Actor (column 2) shows the value by which the policy gradient is scaled. Critic (column 3) lists the algorithm used to compute the state estimates. Reference (column 4) includes the original source where the algorithm was proposed.

A. CACLA

The CACLA algorithm updates the critic using the same update target, G_t^λ, as the AC. However, the actor is only updated if the TDE is positive, i.e. the executed action resulted in a state estimate better than expected. The update equation is also modified to replace the TDE with the amount of exploration that was taken in the executed action:

$$\psi_{t+1} = \psi_t + \alpha_a(a_t - \pi_t(x_t))\frac{\nabla_\psi \pi(u_t|x_t,\psi_t)}{\pi(u_t|x_t,\psi_t)} : \delta_k > 0 \tag{6}$$

where a_t is the action executed, having been sampled from the exploration strategy distribution.

B. Forward AC

In contrast to CACLA, the Forward AC [7] keeps the same actor update equation (see Equ.5), changing the critic's algorithm instead. Forward AC uses Forward TD(λ) to compute the state estimates that are exactly equal to its *"forward view"* algorithm, the K-bounded λ-return [7]. Equ. 7 - 11 provide the recursive update equations required to implement the K-bounded λ-return, where the full Forward AC algorithm can be seen in [7].

$$G_t^{\lambda|K} = G_t^{\lambda|K-1} + (\gamma\lambda)^{K-1}\delta'_{t+K-1} \tag{7}$$

$$G_t^{\lambda|K} = r_{t+1} + \gamma(1-\lambda)\hat{V}_{t+1} + \gamma\lambda G_{t+1}^{\lambda|K-1} \tag{8}$$

$$G_{t+1}^{\lambda|K-1} = \frac{(G_t^{\lambda|K} - \rho_t)}{\gamma\lambda} \tag{9}$$

$$\delta'_t = r_{t+1} + \gamma\hat{V}(x_{t+1}|\theta_t) - \hat{V}(x_t|\theta_{t-1}) \tag{10}$$

$$\rho_t = r_{t+1} + \gamma(1-\lambda)\hat{V}(x_{t+1}) \tag{11}$$

where K is the bounded limit for the n-step return.

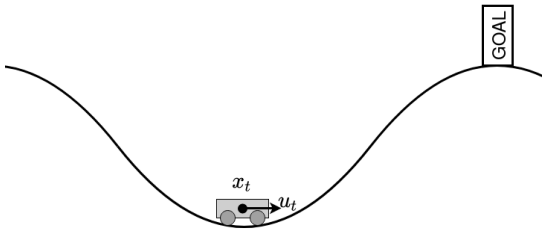

Fig. 2: The MountainCarContinuous-v0 environment. The goal of the environment is for the car to reach the goal position on top of the right mountain. The action, u_t represents a force that is applied to the car. x_t is the state of the car, containing the position on the track and the cars velocity.

C. TOCLA

TOCLA is a novel AC algorithm, that combines Forward AC and CACLA. It uses the K-bounded λ-return as update targets, and hence, implements the Forward TD(λ) as its critic. Updates to the actor, however, use the CACLA update equation (refer to Equ. 6). The combination of these actor and critic updates provide an algorithm ideally suited to continuous-valued real-world applications.

IV. Experiment Setup

There are four software components required to run the RL agents, which obtained the data presented in this paper:

- RL environment framework OpenAI Gym [10]
- Autonomous-Learning-Library (ALL) framework [14]
- ALLAgents GitHub repository [15]
- GEneralised Genetic Algorithm (GEGA) repository [13]

A. OpenAI Gym and Mountain car benchmark

The OpenAI Gym provides a consistent set of benchmark environments that researchers can use to make fairer comparisons to other algorithms [10]. From them, the MountainCarContinuous-v0 environment was selected for our experiments, specifically for its reward function (Equ.12), that can be defined as follows:

$$r_t = \begin{cases} 100 - \sum_{i=0}^{i=t} u_i^2, & \text{if } p > 0.5 \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

where r_t is the reward received at timestep, t, u_i is the action that was taken at timestep i, and p is the position on the track. To solve this environment, an average reward of over 90 must be obtained for 100 consecutive episodes.

This environment is designed not only to reward the agent for reaching the goal state, but also for minimising the force used to reach the goal. Hence, it purposely discourages the learning of bang-bang control responses.

MountainCarContinuous-v0 is a continuous valued variant of the traditional mountain car problem [3]. Fig.2 shows a visual representation of the problem. A car is on a one-dimensional track, positioned between two "mountains", with a state space of $x_t \ \epsilon \ [p, \ \Delta p]$, where p is the position on the track, and Δp is the car velocity. The action space is a one-dimensional continuous variable, $u_t \ \epsilon \ \mathbb{R}$, that represents the driving force the car should apply. A positive force moves the car to the right, whilst a negative force moves the car to the left. The goal is to drive up the mountain on the right. However, the car engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

Parameters passed to GEGA

Description	Parameter	Type
Population size	P	$\mathbb{R}$
Number of generations	T_{max}	$\mathbb{R}$
Minimise fitness	$minimise$	$True \mid False$
Number of genes	g_n	$\mathbb{R}$
Gene bounds	g_b	$[[\mathbb{R}, \mathbb{R}]]^{g_n}$
Gene mutation probability	$g_{mut\ prob}$	$[\mathbb{R}]^{g_n}$
Gene mutation type	$g_{mut\ type}$	$[linear \mid log \mid gaussian]^{g_n}$
Gene absolute tolerance	g_{tol}	$[\mathbb{R}]^{g_n}$
Termination fitness threshold	$T_{threshold}$	$\mathbb{R}$
Termination number of generations	T_{gen}	$\mathbb{R}$

TABLE II: Parameter (column 2) shows the list of variables used by GEGA, along with their description (column 1), and type (column 3). $[\mathbb{R}, \mathbb{R}]$ is a two element vector containing a lower and upper real value, respectively.

B. Learning agent framework

The ALL software framework [14] is an extendable framework used to build and evaluate RL agents which can be run on the integrated OpenAI Gym framework. It also contains a wrapper around the PyTorch torch.nn library [19], enabling the actor and critic approximators to be GPU accelerated. Another benefit of using the PyTorch library is the automatic network gradient calculation that is part of the autograd library.

C. ALLAgents

ALLAgents is a repository that we have developed to extend the ALL framework with the implementations of the AC, CACLA, Forward AC, and TOCLA algorithms that we use in this paper. We have made it freely available in GitHub repository [15].

D. GEneralised Genetic Algorithm

A GA implementation, called GEneralised Genetic Algorithm (GEGA), was developed to optimise the learning hyperparameters of the RL agents in the ALLAgents library. However, GEGAs interface has been kept as generic as possible to allow it to be used for other optimisation problems.

The primary development goal of GEGA was the level of configuration available at the individual parameter (gene) level. Table II lists all of the parameters given to GEGA. These must be defined, and are required in order to enable the algorithm

Algorithm 1: GEGA Evolution Pseudocode

```
initialise population of solutions;
initialise solutions_history;
if load_past_result then
    load population, fitness, solutions_history, and
     generation_number from file
else
    calculate fitness for all solutions in population;
    add_to_solutions_history(population);
end
while not stop_criteria do
    Perform tournament selection of two parents from
     the population (with tour = 4);
    Stochastically perform a cross-over operation to
     generate a child_solution;
    Stochastically apply a mutation to the
     child_solution;
    if child_solution not in solutions_history then
        calculate fitness of child_solution;
        add_to_solutions_history(child_solution);
        if child_fitness better than any parent then
            replace the worst parent solution in
             population with the child;
        else
            discard child solution;
        end
    end
end
```

to find a viable optimal solution in the continuous-valued search-space. The included properties that can be grouped in evolution, gene, and termination. Evolution contains properties such as the population size, number of generations, number of genes in an individual, and whether the goal is to minimise or maximise the fitness. A Gene is defined by the following properties:

- lower and upper bounds for each parameter
- an absolute tolerance that is used to check for similar past solutions
- mutation type and its associated properties (e.g. Gaussian distribution and its standard deviation)

Termination properties indicate to GEGA when the optimisation is completed. This condition is described using three properties. Firstly, a fitness threshold value that the best solution in the population must reach before termination is considered. Secondly, the number of generations that must have passed without any improvement in the best fitness value. If neither of these two termination conditions get satisfied, then GEGA will run until the specified maximum generation is reached.

A generalised description of GEGA's behaviour is as follows. A population, P, of m solutions, S, to be optimised is kept, where each parameter in an individual solution $s \in S$ is referred to as a *gene*. A solution s_i, is defined as:

$$s_i = \langle gene_1,\ gene_2,\ \ldots,\ gene_g \rangle \tag{13}$$

where g is the number of genes in the solution, and each gene $gene_i$, is a real-valued parameter to be optimised. GEGA was designed to work with continuous-valued solutions.

Each solution in the population has an associated fitness value, which represents a numerical measure of how good a solution is. A selection process is then executed which selects two solutions to be evolved from the population as parents. A tournament selection process was chosen as the selection strategy. This selection process selects individuals from the population by randomly selecting individuals from the population to create a '*tournament*', of size 'tour'. The individual with the best fitness from the *tournament* is then chosen, and a new tournament is created. This process is repeated until the desired number of individuals have been chosen. To create an offspring solution, cross-over and mutation genetic operations are applied to the selected parents. After the offspring has its fitness calculated, an elitism replacement strategy is used.

The replacement process inserts the offspring solution into the population in the place of the parent solution with the worst fitness, providing that the offspring fitness is better. As a result, over generations the solutions in the population improve. The optimisation loop, which GEGA executes, is briefly described in Alg.1

GEGA includes a memory-based supporting tool, where before calculating a new offspring's fitness, it checks whether a 'similar' individual has been run in the past. The user of the library defines the conditions for what constitutes 'similar' individuals through an absolute tolerance for each chromosome. The benefits of this mechanism are two-fold. Firstly, executing the fitness function can be very costly in terms of execution time. Hence, if a solution has already been tested, it does not need run again. Secondly, it encourages diversification of solutions, since duplicates can be avoided.

V. Methodology

The comparison of the RL algorithms are presented in two steps:

1) a parameter optimisation procedure. Its goal is to find the optimal hyper-parameter values for each algorithm to ensure fair comparisons among them. Insights into the sensitivity and importance of each parameter's selection will also be gathered from this part.
2) a set of experiments conducted (using the selected hyper-parameter values), which repeats each algorithms' RL process 20 times to obtain statistics of the averaged reward per episode. The goal is to evaluate the performance of each algorithm in terms of the learning capability, speed that the benchmark problem is solved and its consistency.

The remainder of this section describes common aspects to both the optimisation procedure and repeated learning processes. On the selected MountainCarContinuous-v0 environment, a 'learning run' comprises of running the learning

agent on that environment for 150 episodes, each with 1000 timesteps. The result of which is the return vector, R:

$$R = \langle r_1^{accum},\ r_2^{accum},\ \dots\ r_{150}^{accum} \rangle \tag{14}$$

where r_n^{accum} is the accumulated return of the 1000 timesteps for the n^{th} episode. A value of 0.99 was selected for the learning discount factor, γ, for all experiments. A static γ value was chosen to keep the dimensionality of the optimisation space as low as possible. This value is also close to the discount factor used by Veeriah *et al.* in [7], without being a completely undiscounted task (i.e. $\gamma = 1$).

Both of the ANNs in the actor and the critic, respectively, had the following architecture: two input nodes; two hidden layers with 400 and 300 units, respectively, using the rectified linear unit activation function. A Xavier uniform distribution was used for the ANN weight initialisation [20]. From the initial learning rate value, which is tuned in the following section, the ADAM adaptive learning rate optimisation algorithm was used [21].

Both components, the position and velocity of the car, of the state space were normalised by scaling them to between $[-1, 1]$ before being passed to the ANN as input. The output of the critic ANN was the state estimate, and the output of the actor was the deterministic action to take given the current state. The action executed in the environment was sampled from a Normal distribution centred on the deterministic action, to introduce exploration in the environment. The Normal distribution started with a standard deviation of 1.0, which was decayed at a rate of 0.9995^e, where e is the episode number, until a minimum value of 0.1 was reached.

A. Parameter optimisation procedure

The chromosomes in the pool of solutions had the following structure and range of possible values, where applicable, for all of the experiments.

- actor and critic learning rates, α_π and α_c, had initialisation and mutation bounds set to the range $[1e^{-6},\ 1e^{-1}]$
- λ had its initialisation and mutation bounds set to $[0, 1]$
- λ used a linear mutation function

For the AC and CACLA algorithms, only the actor and critic learning rates were optimised. For FAC and TOCLA, the λ parameter was also optimised since this plays an important role of varying the number of steps that state estimates are estimated over, and also varying the delay before the initial learning update [6], [7].

The goal of the hyper-parameter optimisation procedure was to find the values of α_c, α_π, and λ that would maximise the return vector R:

$$\langle\ \alpha_c,\ \alpha_\pi,\ \lambda \rangle = argmax(R) \tag{15}$$

To achieve this, a fitness function had to be designed to quantitatively measure a *learning run* based on R. Equ.16 was chosen as the fitness function. From this equation, it can be seen that the fitness value is calculated from two *learning runs*. The reason for this is to average out some of the stochasticity of the learning process, whilst minimising execution time. This mechanism is called *fitness by approximation* [22]. During each *learning run*, the RL agent is following a stochastic policy to enable it to explore its search space. As a result, the accumulated reward per episode and, hence, R are also stochastic.

$$\text{fitness} = \sum_{i=1}^{i=150} (|B - \frac{\sum_{i=1}^{i=2} R_i}{2}|) \tag{16}$$

where i is the episode number, and B is a vector of return values that represent the best possible reward per episode. Therefore, given Equ.12, a vector of size $\mathbb{R}^{150}$ with values of 100 was chosen as this makes the fitness Equ. 16 calculate the area between the *learning run's* vector of rewards and the maximum reward possible. Hence, minimising this value should be equal to a greater reward, and a better learnt policy. The tournament selection returned two individuals and used a tour value of four.

The optimisation was repeated ten times for each algorithm, where each execution is referred to, henceforth, as an *'optimisation run'*, and each *optimisation run* lasts for 400 generations. The optimum hyper-parameter values were selected by taking the average of the parameters which achieved the lowest fitness score from each of the ten *optimisation runs*.

B. Performance experiment

To compare the AC, CACLA, FAC, and TOCLA algorithms, we assess their resulting averaged reward and standard deviation per episode and time. These metrics demonstrates several properties of the algorithms:

1) the number of episodes taken before solving the environment indicates the rate of learning
2) the steady-state reward after the environment is solved indicates the performance of the learnt policy
3) the standard deviation of the averaged reward indicates the stability of the learnt policy

To obtain the averaged reward per episode, 20 *learning runs* were executed for each algorithm using the hyper-parameter values identified from the optimisation experiments. Hence, given that the setup is exactly the same for all of the *learning runs*, any difference between them is either a result of the algorithm or the stochasticity of the policy (the amount of which was consistent across all of the algorithms).

VI. Results

In the proceeding subsections, we present the performance achieved by AC, CACLA, FAC, and TOCLA on the mountaincar benchmark environment. The first results correspond to the optimal hyper-parameter values selected according to their corresponding fitness scores, having been optimised according to Sect.V-A. The distribution, across the ten *optimisation runs*, of the hyper-parameters and fitness values are also presented.

The second results shows the performance of each algorithm, averaged over 20 *learning runs*, using the optimal hyper-parameter values from the first experiment.

Optimal hyper-parameter values

Algorithm	α_π	α_c	λ
AC	0.0005309958359 ± 0.0005181446729	0.0008687286902 ± 0.0003795707839	
CACLA	0.004333557417 ± 0.003041052691	0.0007373596187 ± 0.0004422380522	
FAC	0.001369217032 ± 0.0008807837024	0.001266006463 ± 0.0003773732798	0.0413122279 ± 0.03605573309
TOCLA	0.00259986294 ± 0.001863315567	0.001125209337 ± 0.001352974874	0.5306405172 ± 0.2657877008

TABLE III: α_π is the actor learning rate, α_c is the critic learning rate, and λ is the multi-step weighting parameter. The mean values were selected as the final hyper-parameters to run the experiments.

A. *Parameter optimisation*

The optimal values for each algorithms α_c, α_π, and λ (where applicable) are shown in Table III, and their distributions in Fig. 3–5. A violin plot is used to show the probability density function of the distribution, whilst a box plot is also included to show the upper and lower bounds, the upper and lower quantiles, the median, and the mean. By observing the shape of the distributions, it will allow us to further extract knowledge of each algorithm's behaviour.

Fig.3 shows the distribution of the actor learning rate (α_π) for AC, CACLA, FAC, and TOCLA, where the spread in the y-axis represents the standard deviation of the distribution. It can, therefore, be seen that CACLA has, by far, the largest standard deviation, followed by TOCLA.

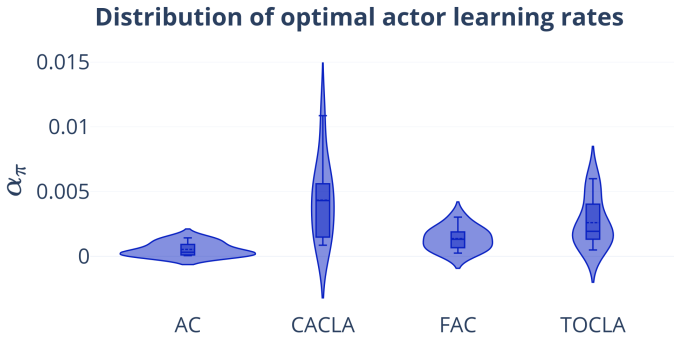

Fig. 3: Distribution of the optimal actor learning rate, calculated over ten *optimisation runs*.

The distribution of α_c is shown in Fig.4. They exhibit significantly less standard deviation across all algorithms than the actor's learning rate. FAC shows the largest standard deviation, with optimal values being spread quite evenly throughout the upper and lower value range. Meanwhile, despite AC and CACLA having a low standard deviation, they both exhibit asymmetrical probability densities.

For the FAC and TOCLA algorithms, the distribution of the optimal λ parameters is shown in Fig.5. This figure highlights a major difference between the algorithms, with the λ parameter for TOCLA ranging between zero and one. Meanwhile, the optimal λ values for FAC are tightly spread around 0.045.

Another important aspect is how the fitness minimisation behaves and the way it reaches convergence, In this regard,

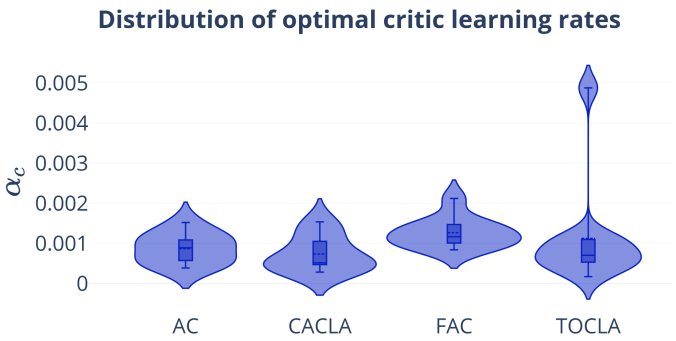

Fig. 4: Distribution of the optimal critic learning rate, calculated over ten *optimisation runs*.

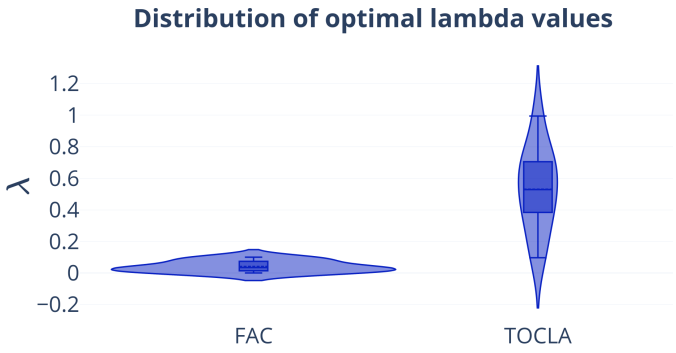

Fig. 5: Distribution of the optimal λ values for the FAC and TOCLA algorithms, calculated over ten *optimisation runs*.

Fig.6 shows the minimisation of the population fitness over 400 generations for AC, CACLA, FAC, and TOCLA. The bars indicate the standard deviation of the fitness, per generation, observed over ten *optimisation runs*. Fig.7 presents the statistics of the achieved fitness, showing that TOCLA obtains the lowest mean value.

Table IV provides more concise numerical values for the mean and standard deviation of the fitness, at the generation intervals $[0,\ 24,\ 49,\ 99,\ 149,\ 199,\ 249,\ 299,\ 349,\ 399]$. The last row in this table is the minimum fitness value that each algorithm achieved. The interval size is decreased below one hundred generations as this region contains the greatest change in fitness.

The algorithms in ascending order of the best fitness values

Sampled average fitness and standard deviation

Generation	AC	CACLA	FAC	TOCLA
0	20542 ± 3832	16728 ± 6929	21030 ± 4412	12083 ± 7514
24	11609 ± 6856	3837 ± 401	5278 ± 3052	3503 ± 673
49	7384 ± 4873	3260 ± 532	3556 ± 1122	2758 ± 321
99	3961 ± 682	2824 ± 189	2731 ± 179	2368 ± 151
149	3755 ± 648	2674 ± 185	2579 ± 163	2333 ± 190
199	3570 ± 371	2649 ± 181	2549 ± 163	2315 ± 192
249	3491 ± 416	2616 ± 202	2507 ± 203	2218 ± 185
299	3405 ± 329	2616 ± 202	2424 ± 140	2217 ± 185
349	3372 ± 330	2602 ± 201	2419 ± 139	2217 ± 185
399	3341 ± 315	2564 ± 229	2419 ± 139	2187 ± 191

TABLE IV: The mean values and standard deviation for the fitness achieved by AC, CACLA, FAC, and TOCLA. Rows are at intervals of 50 generations throughout the maximum number of generations run, except for the second row which is at an interval of 25 generations

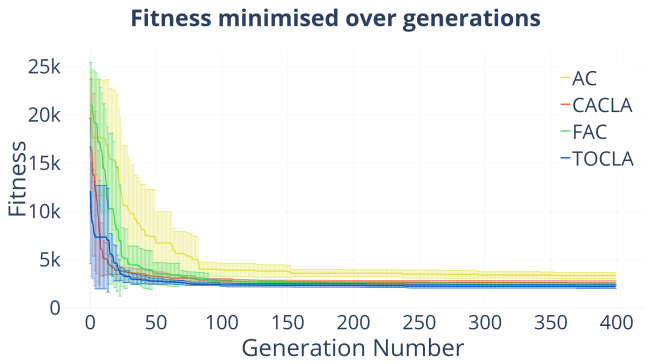

Fig. 6: Fitness minimised over generations for AC, CACLA, FAC, and TOCLA. Calculated over ten *optimisation runs*.

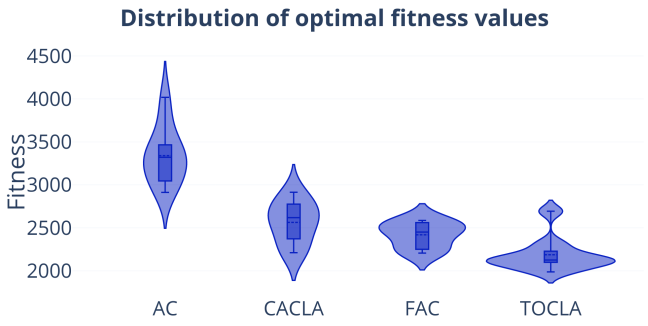

Fig. 7: Distribution of the optimal fitness values for AC, CACLA, FAC, and TOCLA. Calculated over ten *optimisation runs*.

were AC, CACLA, FAC, and TOCLA. AC unquestionably performed worst, whilst there is less difference in the final fitness value between CACLA, FAC, and TOCLA. However, a notable observation throughout the generations is the decreased standard deviation for FAC and TOCLA, which indicates a reduction in the diversity of the population, likely due to their proximity to the global optima. Another interesting aspect is that FAC reached its best fitness value from generation 300, meanwhile the rest of the algorithms continue improving at a slow pace. TOCLA is, in this regard, significantly better, reaching the minimum mean fitness value that FAC achieved in generation 100.

B. Algorithm performance

Using the selected optimal learning parameters from Table III, the accumulated reward per episode over 20 *learning runs* was gathered and the mean and standard deviation calculated. The result of which is shown in Fig.8 and Table V. Fig.8 shows the average, over 20 *learning runs*, accumulated reward and standard deviation per episode for AC, CACLA, FAC, and TOCLA. Table V again shows the accumulated reward and standard deviation per episode for AC, CACLA, FAC, and TOCLA, but it shows quantitative values 10 episode intervals.

The AC algorithm did not perform well with the selected parameters. In Fig.9, the 20 *learning runs* of the AC are shown, having been split into two groups: a successful group, where a learnt policy was able to solve the environment, and a failed group, where the policy saw little improvement or diverged from the optimal. The data shown is the average accumulated reward and standard deviation, per group. Fig.9 aids in demonstrating why the average accumulated reward per episode of AC in Fig.8 is so low, and with such a high standard deviation. From the 20 *learning runs*, the AC only learnt to solve the environment 8 times, where 60% of *learning runs* failed to improve the policy from its initial state. All of the remaining algorithms consistently solved the environment. A noteworthy improvement over AC is achieved by CACLA, as simply by changing the actor update equation, it substantially improved its stability. Compared with FAC and TOCLA, however, CACLA was slower to solve the environment, requiring approximately double the number of episodes that FAC and TOCLA needed.

From Fig.8 it can be observed that there is little difference

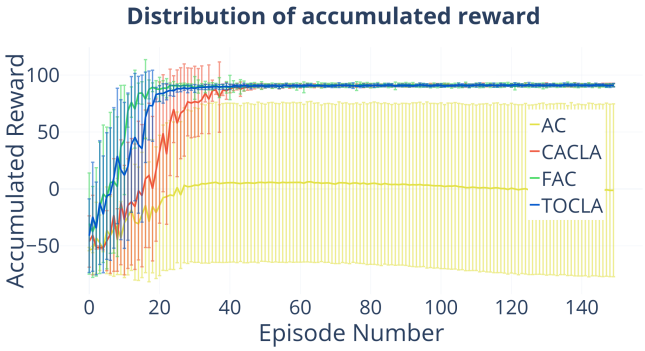

Fig. 8: Accumulated reward per episode, calculated over 20 *learning runs*, whilst using the selected learning parameters from Table III

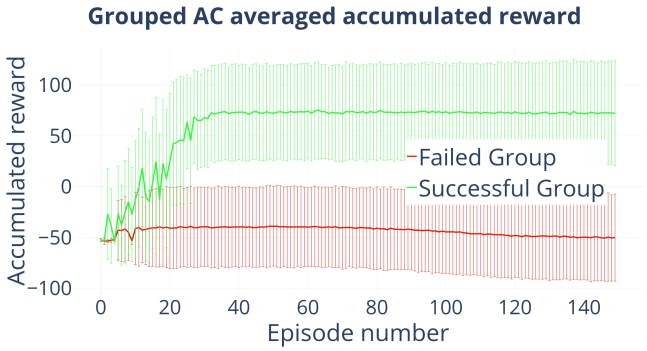

Fig. 9: Accumulated reward per episode of the AC, grouped into successful and failed groups (with respect to solving the environment), whilst using the selected learning parameters from Table V.

between FAC and TOCLA, with FAC solving the environment marginally before TOCLA. However, having solved the environment (i.e. after episode 40), it is clearly visible from Table.V that TOCLA and CACLA have a reduced standard deviation over FAC, which means that learning has effectively stopped. In Fig.8, an observation is made that despite TOCLA achieving a lower fitness (on average) than FAC, it appears to learn slightly slower than FAC. This is counter-intuitive and shall be discussed in the following section.

VII. Discussion

In Section VI, we presented the results of comparing the AC, CACLA, FAC, and TOCLA algorithms on the MountainCarContinuous-v0 benchmark. Observed from these results was that TOCLA, on average, achieved a more optimal policy during the optimisation procedure than the rest of the AC algorithms. However, in the performance experiment, TOCLA did not have the best policy, which belonged to FAC. This, as well as other interesting observations from our results are now discussed.

The poor performance of AC is understandable, given its one-step update target and the benchmark environment's reward function. The reward function only returns a non-zero value if the goal has been reached, effectively rewarding algorithms that can propagate experienced reward backward through time.

In contrast, despite CACLA using the same one-step TD algorithm for its critic as the AC algorithm, it achieved a significantly more stable performance, solving the environment every time. This improvement should originate from the actor update equation, given that it is the only difference between AC and CACLA. This replaces the TDE with the amount of stochastic exploration that was taken. We hypothesise that this result is due to one of CACLA's main update ideologies, which states that moving the actor's output in the opposite direction from the last action when experiencing a negative TDE is not necessarily correct, and may, in fact, be detrimental. This hypothesis is influenced by the significantly larger standard deviation of α_π, observed in Fig.3, compared with AC, FAC, and TOCLA. Given that CACLA's optimal fitness standard deviation in Fig.7 is similar to AC, with such a large standard deviation in CACLA's α_π, this suggests that α_π for CACLA is significantly less sensitive than the AC's. It, therefore, makes CACLA more robust, all from removing the TDE from the actor update equation.

FAC also achieved a large improvement over the standard AC, by using multi-step update targets, specifically designed for applying an ANN to approximate the critic. Given that the optimal λ for FAC was found to be 0.045, it would appear that the Forward TD(λ) may converge to realistic state estimates, significantly faster than the standard TD algorithm. In effect, as FAC shares the same actor update as AC, it must be obtaining realistic state estimates faster, which reduces the time at the start of learning, where the TDE is wrong and is adversely affecting the actor as a result.

TOCLA was founded on the ideal that utilising the state-of-the-art actor and critic technologies from the CACLA update and Forward TD(λ), respectively, it would result in a bleeding edge algorithm. During the optimisation experiment, from Table IV, it can be seen that this ideal looked promising, with TOCLA demonstrating a more optimal policy with a more even distribution. However, in Fig.8, it was surprising to see that FAC actually reached the optimal policy sooner than TOCLA. Once converged to the optimal policy, looking at Table V, it can be seen that the standard deviation of TOCLA is less than FAC, indicating stability. The reason for this is two-fold: firstly, the exploration is decayed with the number of episodes, and secondly, the CACLA actor update does not update the policy for $\delta_t <= 0$. We hypothesise that the explanation for the slower rate of convergence of TOCLA lies in the decaying of exploration, as this is directly reducing the step size of the policy update. Also, however, the very large standard deviation of the λ parameter for TOCLA (Fig.5), the method in which the optimal values were selected, and our *optimisation run* sample size could also be contributing factors. We speculate that, with more *optimisation runs* and clustering of the resulting λ values, and decreasing the exploration decay rate, that the TOCLA algorithm may achieve the optimal policy.

Our results show a valuable insight into the interaction between actor and critic modules and the effect that substituting

Sampled accumulated reward and standard deviation

Generation	AC	CACLA	FAC	TOCLA
0	-53 ± 1	-46 ± 23	-30 ± 44	-41 ± 32
9	-43 ± 32	-12 ± 55	37 ± 54	17 ± 61
19	-21 ± 58	17 ± 63	88 ± 7	83 ± 6
29	2 ± 66	75 ± 31	90 ± 2	89 ± 2
39	5 ± 70	86 ± 7	89 ± 5	90 ± 3
49	6 ± 69	90 ± 2	91 ± 3	91 ± 1
59	6 ± 70	91 ± 1	90 ± 3	91 ± 1
69	5 ± 70	91 ± 1	91 ± 2	90 ± 1
79	4 ± 71	91 ± 1	91 ± 2	91 ± 1
89	4 ± 71	91 ± 1	91 ± 2	91 ± 1
99	3 ± 72	91 ± 1	91 ± 2	91 ± 1

TABLE V: The averaged accumulated reward and standard deviation for AC, CACLA, FAC, and TOCLA across 20 *learning runs*. Rows are sampled every 10 episodes throughout the first 100 episodes

just the actor or critic from one algorithm to another can have. This kind of research is invaluable when the resulting algorithms and their behaviours even in benchmark problems are non-trivial, with non-intuitive outcomes possible.

VIII. Conclusion

Reinforcement learning provides a general framework for solving problems modelled as a MDP. This makes reinforcement learning immensely applicable to many real-world problems. However, these problems often occur in continuous-valued domains.

We proposed a new algorithm called TOCLA after recent work brought focus onto incremental online temporal difference learning methods for non-linear function approximation. TOCLA was compared against the original AC, CACLA, and FAC on the MountainCarContinuous-v0 benchmark of OpenAI Gym. A GA implementation called GEGA was developed to tune the learning rate and lambda hyper-parameter values for the comparison. The hyper-parameter value optimisation procedure using GEGA resulted in TOCLA performing best of the four actor critic algorithms. Using the selected optimal hyper-parameter values and repeating the learning process, however, resulted in the FAC performing best. This unforeseen result lead to some insights into the effects of the different components, such as the temporal difference error, exploration, and multi-step update targets. All of the source code used to gather the experiment data in this paper is freely available on GitHub, including GEGA, and implementations of all four of the AC algorithms.

For future work, multiple tracts can be taken. Firstly, the GEGA library can be improved to include alternative selection, cross-over, and mutation strategies. Secondly, different *fitness by approximation* approaches can also be investigated to see their effect. Finally, the performance of TOCLA will be evaluated on other control applications, including the control of an autonomous underwater vehicle.

References

[1] Thor I Fossen. *Guidance and control of ocean vehicles*. John Wiley & Sons Inc, 1994.

[2] J. Katz and A. Plotkin. *Low-speed aerodynamics*, volume 13. Cambridge university press, 2001.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[4] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[5] H.V Seijen and R. Sutton. True online td (lambda). In *Proceedings of the 31st International Conference on Machine Learning*, pages 692–700, 2014.

[6] Harm van Seijen. Effective multi-step temporal-difference learning for non-linear function approximation. *arXiv preprint arXiv:1608.05151*, 2016.

[7] V. Veeriah, H. van Seijen, and R.S Sutton. Forward actor-critic for non-linear function approximation in reinforcement learning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 556–564, 2017.

[8] Hado Van Hasselt and Marco A Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 272–279. IEEE, 2007.

[9] S.A. Fjerdingen, E. Kyrkjeboe, and A.A. Transeth. Auv pipeline following using reinforcement learning. In *41st International Symposium on Robotics (ISR) and 6th German Conference on Robotics (ROBOTIK)*, pages 1–8, June 2010.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[11] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[12] Pawel Cichosz. Truncating temporal differences: On the efficient implementation of TD (lambda) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318, 1994.

[13] Generalised genetic algorithm (gega) github repository. https://github.com/gordon-frost-hwu/gega. Accessed: 2020-07-05.

[14] Autonomous learning library (all) reinforcement learning framework. https://github.com/cpnota/autonomous-learning-library. Accessed: 2020-03-10.

[15] Allagents - an extension to the autonomous learning library with forward td(λ) actor-critic agents. https://github.com/gordon-frost-hwu/ALLAgents. Accessed: 2020-06-08.

[16] Justin Boyan and Andrew W Moore. Generalization in reinforcement learning: Safely approximating the value function. *Advances in neural information processing systems*, pages 369–376, 1995.

[17] S.P Singh and R.S Sutton. Reinforcement learning with replacing eligibility traces. *Recent Advances in Reinforcement Learning*, pages 123–158, 1996.

[18] Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, and M. et al. Bradbury. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] M. Vallejo and D.W Corne. Evolutionary algorithms under noise and uncertainty: a location-allocation case study. In *IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–10. IEEE, 2016.